THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Java Memory Model Examples: Good, Bad and Ugly

Citation for published version:
Aspinall, D & Ševík, J 2007, Java Memory Model Examples: Good, Bad and Ugly. in *Proceedings of Verification and Analysis of Multi-Threaded Java-Like Programs (VAMP 2007).*

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of Verification and Analysis of Multi-Threaded Java-Like Programs (VAMP 2007)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 04. May. 2023

Java Memory Model Examples: Good, Bad and Ugly

David Aspinall and Jaroslav Ševčík

August 8, 2007

Abstract

We review a number of illustrative example programs for the Java Memory Model (JMM) [6, 3], relating them to the original design goals and giving intuitive explanations (which can be made precise). We consider good, bad and ugly examples. The good examples are allowed behaviours in the JMM, showing possibilities for non sequentially consistent executions and reordering optimisations. The bad examples are prohibited behaviours, which are clearly ruled out by the JMM. The ugly examples are most interesting: these are tricky cases which illustrate some problem areas for the current formulation of the memory model, where the anticipated design goals are not met. For some of these we mention possible fixes, drawing on knowledge we gained while formalising the memory model in the theorem prover Isabelle [1].

1 Introduction

The Java Memory Model (JMM) [6, 3] is a relaxed memory model which acts as a contract between Java programmers, compiler writers and JVM implementors. It explains possible and impossible behaviours for multi-threaded programs. The JMM is necessary to allow efficient execution and compilation of Java programs, which may result in optimisations that affect the order of memory operations. The case that is usually desirable is *sequential consistency* (SC) [4], which, roughly speaking, says that the outcome of executing a multi-threaded program should be equivalent to the outcome of executing some sequential ordering of its operations which agrees with the order that statements appear in the program. Sequential consistency has acted as a correctness criterion in the study of relaxed memory models and numerous variations have been explored; we discuss our own precise definition of SC for the JMM later.

Sequential consistency helps make multi-threaded programming comprehensible to the programmer. But for parts of a program which are executing in unrelated threads, sequential consistency may not be required. Moreover, in pursuit of the best possible performances, sophisticated concurrent algorithms have been designed which work in the presence of data races. A *data race* in a program is a point where the program itself fails to specify an ordering on conflicting actions across threads, so sequential consistency is underdefined.

The JMM is one of the first explored memory models which connects a high-level programming language to low-level executions (most other relaxed memory

models work at the hardware level). The JMM has been designed to make three guarantees:

1. *A promise for programmers*: sequential consistency must be sacrificed to allow optimisations, but it will still hold for data race free programs. This is the data race free (DRF) guarantee.

2. *A promise for security*: even for programs with data races, values should not appear "out of thin air", preventing unintended information leakage.

3. *A promise for compilers*: common hardware and software optimisations should be allowed as far as possible without violating the first two requirements.

The key question one asks about a program is whether a certain outcome is possible through some execution. Unfortunately, with the present JMM, it can be quite difficult to tell the answer to this! In part, the memory model has been designed around a set of examples (e.g., the causality tests in [8]) which have helped shape the definitions; but gaps and informality in the present definitions mean that there are still unclear cases.

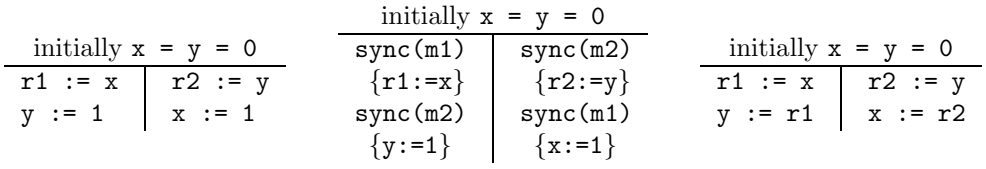
Points 1 and 2 act to *prohibit* certain executions, whereas point 3 acts to *require* certain executions. It seems that only point 1 provides a precise set of behaviours that are disallowed, i.e., the non-SC behaviours for data race free programs. Regarding point 3, exactly which optimisations must be allowed has been a source of some debate and is still in flux [8, 9]. Regarding point 2, the "out of thin air" requirement has yet to be precisely characterised; we only know of forbidden examples which violate causality to allow behaviours that result in arbitrary values.

This paper discusses some illustrative examples for the Java Memory Model, relating them back to these goals and to the JMM definitions, and in particular, the definitions as we have formalised them [1]. Our contribution is to collect together some canonical examples (including tricky cases), and to explain how they are dealt with by our formal definitions, which represent an improvement and clarification of the official definitions. Despite the intricacies of the JMM definitions, we present the examples at an informal level as far as possible. We also give some opinions on future improvements of the JMM.

The rest of this paper is structured as follows. Section 2 explains intuitively how behaviours are justified in the memory model. Sections 3, 4 and 5 then present the examples: the good (allowed), the bad (prohibited) and the ugly (tricky cases where there is disparity between the JMM design aims and the actual definitions). Appendix A recalls some of the definitions of the JMM in a more precise format for reference; it should be studied by those who seek a complete understanding but can be ignored by a casual reader. Section 6 concludes.

2 A Bluffer's Guide to the JMM

Motivation. Before we introduce the memory model, let us examine three canonical examples (from [6]), given in Fig. 1, which illustrate the requirements mentioned above. The programs show statements in parallel threads, operating on thread-local registers (`r1`, `r2`, ...) and shared memory locations (`x`, `y`, ...).



A. (allowed) B. (prohibited) C. (prohibited)

Is it possible to get `r1` = `r2` = 1 at the end of an execution?

Figure 1: Examples of legal and illegal executions.

In an interleaved semantics, program A could not result in `r1` = `r2` = 1, because one of the statements `r1:=x`, `r2:=y` must be executed first, thus either `r1` or `r2` must be 0. However, current hardware can, and often does, execute instructions out of order. Imagine a scenario where the read `r1:=x` is too slow because of cache management. The processor can realise that the next statement `y:=1` is independent of the read, and instead of waiting for the read it performs the write. The second thread then might execute both of its instructions, seeing the write `y:=1` (so `r2` = 1). Finally, the postponed read of `x` can see the value 1 written by the second thread, resulting in `r1` = `r2` = 1. Similar non-intuitive behaviours could result from simple compiler optimisations, such as common subexpression elimination. The performance impact of disabling these optimisations on the current architectures would be huge; therefore, we need a memory model that allows these behaviours.

However, there are limits on the optimisations allowed—if the programmer synchronises properly, e.g., by guarding each access to a field by a synchronized section on a designated monitor, then the program should only have sequentially consistent behaviours. This is why the behaviour `r1` = `r2` = 1 must be prohibited in program B of Fig. 1.

Even if a program contains data races, there must be some security guarantees. Program C in Fig. 1 illustrates an unwanted "out-of-thin-air" behaviour—if a value does not occur anywhere in the program, it should not be read in any execution of the program. The out-of-thin-air behaviours could cause security leaks, because references to objects from possibly confidential parts of program could suddenly appear as a result of a self-justifying data race. This might let an applet on a web page to see possibly sensitive information, or, even worse, get a reference to an object that allows unprotected access to the host computer.

JMM framework. Now we introduce the key concepts behind the JMM. Unlike interleaved semantics, the Java Memory Model has no explicit global ordering of all actions by time consistent with each thread's perception of time, and has no global store. Instead, executions are described in terms of memory related actions, partial orders on these actions, and a visibility function that assigns a write action to each read action.

An action is a tuple consisting of a *thread identifier*, an *action kind*, and a *unique identifier*. The action kind can be either a normal read from variable x, a normal write to x, a volatile read from v or write to v, a lock of monitor m, or an unlock of monitor m. The volatile read/write and lock/unlock actions

are called *synchronisation actions*. An *execution* consists of a *set of actions*, a *program order*, a *synchronisation order*, a *write-seen* function, and a *value-written* function. The program order ($\leq_{po}$) is a total order on the actions of each thread, but it does not relate actions of different threads. All synchronisation actions are totally ordered by the synchronisation order ($\leq_{so}$). From these two orders we construct a happens-before order of the execution: action a happens-before action b ($a \leq_{hb} b$) if (1) a synchronises-with b, i.e., $a \leq_{so} b$, a is an unlock of m and b is a lock of m, or a is a volatile write to v and b is a volatile read from v, or (2) $a \leq_{po} b$, or (3) there is an action c such that $a \leq_{hb} c \leq_{hb} b$. The happens-before order is an upper bound on the visibility of writes—a read happening before a write should never see that write, and a read r should not see a write w if there is another write happening "in-between", i.e., if $w \leq_{hb} w' \leq_{hb} r$ and $w \neq w'$, then r cannot see w.[1]

We say that an execution is *sequentially consistent* if there is a total order consistent with the program order, such that each read sees the most recent write to the same variable in that order. A pair of memory accesses to the same variable is called a *data race* if at least one of the accesses is a write and they are not ordered by the happens-before order. A program is *correctly synchronised* (or *data-race-free*) if no sequentially consistent execution contains a data race.

A tricky issue is initialisation of variables. The JMM says

> The write of the default value (zero, false, or null) to each variable synchronises-with to the first action in every thread [7]

However, normal writes are not synchronisation actions and synchronises-with only relates synchronisation actions, so normal writes cannot synchronise-with any action. For this paper, we will assume that all default writes are executed in a special initialisation thread and the thread is finished before all other threads start. Even this interpretation has problems; we mention them in Sect. 5.

Committing semantics. The basic building blocks are *well-behaved* executions, in which reads are only allowed to see writes that happen before them. In these executions, reads cannot see writes through data races, and threads can only communicate through synchronisation. For example, programs A and C in Fig. 1 have just one such execution—the one, where $\texttt{r1} = \texttt{r2} = 0$. On the other hand, the behaviours of program B are exactly the behaviours that could be observed by the interleaved semantics, i.e. $\texttt{r1} = \texttt{r2} = 0$, or $\texttt{r1} = 1$ and $\texttt{r2} = 0$, or $\texttt{r1} = 0$ and $\texttt{r2} = 1$. In fact, if a program is correctly synchronised then its execution is well-behaved if and only if it is sequentially consistent. This does not hold for incorrectly synchronised programs, see Sect. 5.

The Java Memory Model starts from a well-behaved execution and *commits* one or more data races from the well-behaved execution. After committing the actions involved in the data races it "restarts" the execution, but this time it must execute the committed races. This means that each read in the execution must be either committed and see the value through the race, or it must see the write that happens-before it. The JMM can repeat the process, i.e., it may choose some non-committed reads involved in a data race, commit the writes involved in these data races if they are not committed already, commit the

[1] For details, see Defs 2, 4 and 6 in App. A.

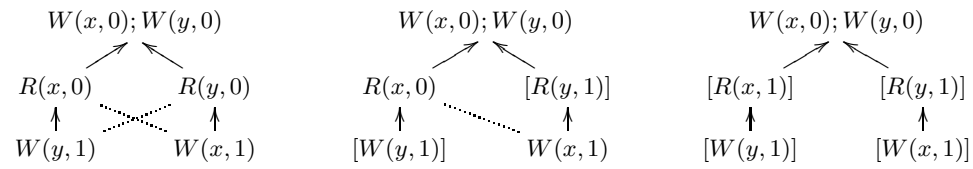


Figure 2: Justifying executions of program A from Fig. 1.

chosen reads, and restart the execution. The JMM requires all of the subsequent executions to preserve happens-before ordering of the committed actions.

This committing semantics imposes a causality order on races—the outcome of a race must be explained in terms of previously committed races. This prevents causality loops, where the outcome of a race depends on the outcome of the very same race, e.g., the outcome `r1` = 1 in program C in Fig. 1. The DRF guarantee is a simple consequence of this procedure. If there are no data races in the program, there is nothing to commit, and we can only generate well-behaved executions, which are sequentially consistent. In fact, the JMM actually commits all actions in an execution, but committing a read that sees a write that happens before it does not create any opportunities for committing new races, because reads must see writes that happen-before them in any well-behaved execution. Therefore the central issue is committing races, and we explain our examples using this observation.

3 Good executions

The interesting executions are those which are not sequentially consistent, but are legal under the JMM.

Simple reordering. First, we demonstrate the committing semantics on program A in Fig. 1. In the well-behaved execution of this program, illustrated by the first diagram in Fig. 2, the reads of `x` and `y` can only see the default writes of 0, which results in `r1` = `r2` = 0.

There are two data races in this execution (depicted by the dotted lines, the solid lines represent the happens-before order)—one on `x` and one on `y`. We can commit either one of the races or both of them. Suppose we commit the race on `y`. In the second diagram we show the only execution that uses this data race; the committed actions are in brackets and the committed read sees the value of (the write in) the data race. The non-committed read sees the write that happens-before it, i.e., the default write. This execution gives the result `r1` = 0 and `r2` = 1. The JMM can again decide to commit a data race from the execution. There is only one such data race. Committing the data race on `x` gives the last diagram, and results in `r1` = `r2` = 1.

Complex optimisation. The program

initially `x = y = 0`

`r1 := x`	`r2 := y`
`if (r1 != 0)`	`x := r2`
`  y := r1`	
`else`	
`  y := 1`	

,

can result in `r1` = `r2` = 1 using a committing sequence that is very similar to the previous program. First we commit the race on `y` and then the race on `x` with value 1. Note that the actions are not tied to instructions in any way—the write to `y` is first committed using the `else` branch, but it is executed by the `then` branch in the final execution.

Although it might seem counter-intuitive, this behaviour could be the outcome of a compiler optimisation. The compiler may realise that the only values in the program are 0 and 1; that the first thread can never write 0 to `y`; thus it must write 1. As a result, it could replace the `if` statement with an assignment `y:=1`. Then it could reorder this with the statement `r1:=x`, giving the program:

`y := 1`	`r2 := y`
`r1 := x`	`x := r2`

After the reordering we can get `r1` = `r2` = 1 from a SC execution.

4 Bad executions

The JMM aims to prohibit out-of-thin-air behaviours, which are usually demonstrated by program C from Fig. 1.

It contains a causality loop, which potentially could explain any value being written and seen by `r1`. But the JMM prohibits the result `r1` = 1, for example, because we can only commit a race with the value 0. Even after restarting, all reads must see the value 0, because the non-committed ones can only see the default writes, and the committed ones were committed with value 0.

Another powerful tool for determining (in)validity of executions is the DRF guarantee. For example, this program (from [6]):

initially `x = y = 0`

`r1 := x`	`r2 := y`
`if (r1>0) y := 42`	`if (r2>0) x := 42`

is data race free and thus it cannot be that `r1` = 42, because the writes of 42 do not occur in any sequentially consistent execution.

5 Ugly executions

Here we give examples that either show a bug in the JMM, i.e., behaviours that should be allowed but they are not, or behaviours that are surprising.

1. Reordering of independent statements. The first example (from [2]) demonstrates that it is not the case that any independent statements can be reordered in the JMM[2] without changing the program's behaviour:

[2]In [1], we suggest a weakening of the legality definition that fixes this problem while preserving the DRF guarantee.

```
                x = y = z = 0
r1:=z                          | r2:=x
if (r1==1) {x:=1; y:=1}        | r3:=y
       else {y:=1; x:=1}       | if (r2==1 && r3==1)
                               |   z:=1
```

Can we get `r1` = `r2` = `r3` = 1? This requires that each register read sees a write of 1. The only way to commit a data race on `z` with value 1 is to commit the races on `x` and `y` with values 1, so that the write `z:=1` is executed. Note that the writes to `x` and `y` are committed in the `else` branch, and the write to `y` happens-before the write to `x`. However, once we commit the data race on `z`, the first thread must execute the `then` branch in the restarted execution, which violates the requirement on preservation of ordering of the committed actions. So this outcome is impossible.

If we swap the statements in either branch of the `if` statement so they are the same, we *can* justify the result `r1` = `r2` = `r3` = 1. This demonstrates that independent instruction reordering introduces a new behaviour of the program, and so is not a legal transformation in general. This falsifies Theorem 1 of [6].

2. Reordering with external actions. Another counterexample to Theorem 1 of [6] shows that normal statements cannot be reordered with "external statements" (i.e. those which generate external actions such as I/O operations).

```
          x = y = 0
r1:=y                          | r2:=x
if (r1==1)                     | y:=r2
  x:=1                         |
else {print "!"; x:=1}         |
```

Here, the result `r1` = `r2` = 1 is not possible, because we must commit the printing before committing the data race on `x`[3]. However, after swapping the print with `x:=1` in the else branch, we can get the value 1 in both the registers by committing the race on `x` followed by committing the race on `y`. As a result, this reordering is not legal. A fix to this problem has not been proposed yet.[4]

3. Roach motel semantics. A desirable property of the memory model is that adding synchronisation to a program could introduce deadlock, but not any other new behaviour. A special case is "roach motel semantics" ([7]), i.e., moving normal memory accesses inside synchronised regions. Another case is making a variable volatile. We now show examples falsifying the claim that the JMM ensures this property.

First, note that this program

[3] By rule 9 of legality, Def. 7 in the Appendix.

[4] The apparent fix is to drop rule 9; the consequences of this are unknown, although DRF will be preserved.

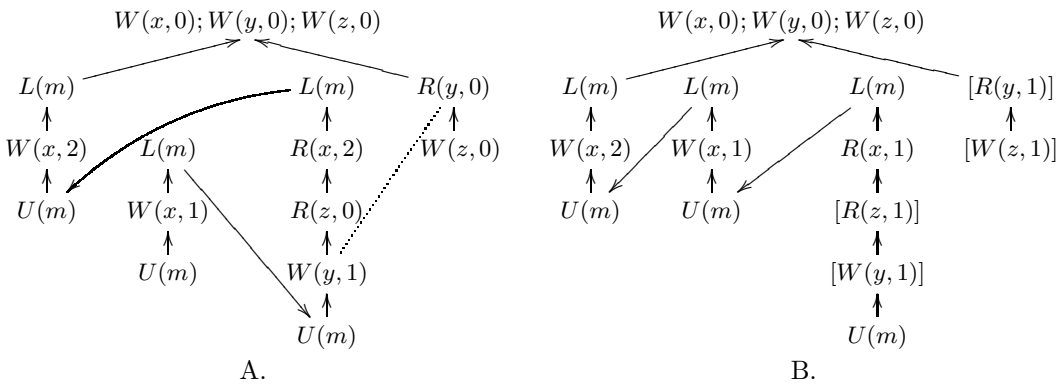


Figure 3: Justifying and final executions for the roach motel semantics counterexample.

initially `x = y = z = 0`

`lock m`	`lock m`	`r1 := x`	`r3 := y`
`x := 2`	`x := 1`	`lock m`	`z :  = r3`
`unlock m`	`unlock m`	`r2 := z`	
		`if (r1 == 2)`	
		`  y := 1`	
		`else`	
		`  y := r2`	
		`unlock m`	

cannot result in `r1` = `r2` = `r3` = 1, because the only way to get a write of 1 into `y` is to commit the data race on `x` with value 2. Once we commit the read of 2 from `x`, the read of `x` must always see value 2 and the register `r1` must be 2 in any subsequent execution, so we cannot have `r1` = 1 in the final execution.

However, if we move the assignment `r1:=x` inside the synchronised block, we can construct an execution where `x:=2` happens-before `r1:=x` using the synchronisation on `m`, and `r1:=x` sees value 2 (see execution A from Fig. 3). From this execution, we can commit the data race on `y` with value 1 without committing the read of `x`. Then we can restart and commit the data race on `z` with value 1, and after another restart of the execution, we let the read of `x` see value 1 (execution B from Fig. 3). As a result, we introduce a new behaviour `r1` = `r2` = `r3` = 1 after moving a normal access into a synchronised block.

Using the same reasoning, if `x` is volatile, we can also get `r1` = `r2` = `r3` = 1 even if `r1:=x` is outside the synchronised block. This demonstrates that making a variable volatile can introduce new behaviours other than deadlock.

4. Reads affect behaviours. In the JMM, reads can affect an execution in perhaps unexpected ways. Unlike an interleaved semantics, removing a redundant read from a program, such as replacing `r:=x;r:=1` by `r:=1` can decrease observable behaviours, because the read of `x` might have been previously committed and it must be used. As a result, introducing redundant reads to a program is not a legal program transformation.

To demonstrate this, consider the program

```
          x = y = z = 0
r1 := z                    | x := 1
if (r1==0)                 | r2 := y
{ r3 := x                  | z := r2
  if (r3==1) y := 1 }      |
else {  r4 := x            |
        r4 := 1            |
        y := r1  }         |
```

and the outcome r1 = r2 = 1. This is a possible behaviour of the program—we can commit the race on x between r3:=x and x:=1, then the race on y between y:=1 and r2:=y, and finally the race on z with value 1 to get the result. But if we remove the (redundant) read r4:=x, we cannot keep the committed race on x after we commit the race on z with value 1.

5. Causality tests. In [8], the causality tests 17–20 are wrong—they are not allowed in the JMM contrary to the claim. We illustrate the problem on causality test 17 (all tests are an instance of one problem and we suggest a fix in [1]):

```
        x = y = 0
r3 := x          | r2 := y
if (r3 != 42)    | x := r2
   x := 42       |
r1 := x          |
y := r1          |
```

The causality test cases state that r1 = r2 = r3 = 42 should be allowed, because the compiler might realize that no matter what is the initial value of variable x in the first thread, r1 will always be 42, and it can replace the assignment by r1 := 42. Then we can get the desired outcome by simple reordering of independent statements. However, there is a subtle bug in the memory model and it does not allow this behaviour (see [1] for more details). This is because rule 7 of legality (Def. 7) is too strong—it requires the reads being committed to see already committed writes in the justifying execution.

One of the causality tests also answers an interesting question: can we always commit actions one-by-one, i.e., each commit set having one more element than the previous one? The answer is negative, as illustrated by the causality test 2:

```
        x = y = 0
r1 := x          | r3 := y
r2 := x          | x := r3
if (r1 == r2)    |
   y := 1        |
```

To get r1 = r2 = r3 = 1, we must commit both reads of x at the same time.

6. Sequential consistency and lock exclusivity. Our next example execution is considered sequentially consistent by the JMM but does not reflect an interleaved semantics respecting mutual exclusion of locks. The JMM says that an execution is sequentially consistent if there is a total order consistent with

the execution's program order such that each read sees the most recent write in that order. In [1], we show that this does not capture interleaved semantics with exclusive locks, demonstrated in this program:

initially x = y = z = 0		
r1 := y	lock m	lock m
x := r1	r2 := x	y := 1
	z := 1	r3 := z
	unlock m	unlock m

By the JMM, it is possible to get r1 = r2 = r3 = 1, using the total order lock m, lock m, y:=1, r1:=y, x:=r1, r2:=x, z:=1, r3:=z, unlock m, unlock m. It is not hard to show that this the only order of reads and writes that is consistent with the program order and the reads see only the most recent writes in that order. However, this order does not respect mutual exclusion of locks. We believe, therefore, that sequential consistency ought to require existence of a total order consistent with both the program order and the synchronisation order. In [1], we have proved that the DRF guarantee also holds for this definition of SC.

7. Default writes and infinite executions. The writes of default values to variables introduce inconsistencies and surprising behaviours in the JMM. We noted earlier that the definition of the happens-before relation is flawed when default writes are considered carefully. Possible fixes, either by making default writes into synchronisation actions, or by forcing the initialisation thread to finish before all other threads start, conflict with infinite executions and observable behaviours in a subtle way.

For example, any infinite execution of the program (from [1]):

```
while(true) { new Object() { public volatile int f; }.f++; }
```

must initialize an infinite number of volatile variables before the start of the thread executing the while loop. Then the synchronisation order is not an omega order, which violates the first well-formedness requirement. So the program above cannot have any well-formed execution in Java.

A related problem arises with the notions of observable behaviour and "hung" program given in [7]: in the program that precedes the above loop with an external action, there can only be a finite observable behaviour but the criteria for being "hung" are not met. Because of this, we suggest to restrict the executions to finite ones, and exclude the default write actions from observable behaviours.

8. Well-behaved executions and SC. For justifying legal executions, the JMM uses "well-behaved executions" [6, Section 1.2]. These are executions with certain constraints, in particular, that each read sees a most recent write in the happens-before order. It might be interesting to examine the relationship between the well-behaved executions and the sequentially consistent executions.

Lemma 2 of [6] says that for correctly synchronised programs, an execution is sequentially consistent if and only if it is well-behaved.

However, the following two examples show that these two notions are incomparable for general programs. First, consider the program

x = 0	
x := 1	r1 := x

and its execution, where `r1` = 1. This is sequentially consistent, but not well-behaved.

On the other hand, the program

`lock m1`	`lock m2`
`y := 1`	`y := 2`
`x := 1`	`x := 2`
`unlock m1`	`unlock m2`
`lock m2`	`lock m1`
`r1 := x`	`r2 := x`
`r3 := y`	`r4 := y`
`unlock m2`	`unlock m1`

has a well-behaved execution, where `r1` = `r4` = 1 and `r2` = `r3` = 2. This result is not possible in any SC execution.

6 Conclusions

We have collected together and explained a set of typical examples for the Java Memory Model, including those good and bad programs used to motivate the definitions, and a set of ugly programs which cause problems for the present JMM definitions. Our aim was to summarise the status of the current JMM, as we understand it, and at the same time make some of the technicalities more accessible than they are in other accounts.

We have explained the examples at a slightly informal level, giving an intuitive description of the checking process one can use to explain why some example is possible (by explaining the commit sequence), or argue why some example is impossible (by explaining that no commit sequence with the desired outcome is possible). We explained the commit sequences in terms of what happens for data races, which is the essential part of the process; this fact is mentioned in Manson's thesis [5] but not easily gleaned from the text or definitions in other published accounts [6, 3].

The ugly cases are the most interesting; clearly something must be done about them to gain a consistent account. The first hope is that the JMM definitions can be "tweaked" to fix everything. We have suggested several tweaks so far. But, because the definitions have some *ad hoc* aspects justified by examples on a case-by-case basis, it isn't clear that it is possible to revise the definitions to meet the case-by-case examples of allowed and prohibited examples, while at the same time satisfy the global property of enabling all easily detectable optimisations of certain kinds. Further examination is required, but it may be that beginning from more uniformly derived semantic explanations (such as the approach of Cenciarelli et al [2]) is a better approach for achieving a more tractable definition.

Although we (naturally!) believe that our own explanations of the examples are accurate, the definitions are quite intricate and other people have made mistakes — perhaps some of the corner cases (such as the causality tests) were valid in previous versions of the model but became broken as it evolved. Therefore it is an obvious desire to have a way to check examples formally and automatically against memory model definitions; in future work we plan to investigate model checking techniques for doing this.

Related work. We have cited the source of examples and made some comparisons in the text; the papers in the bibliography also contain ample further pointers into the wider literature on memory models. The key starting point for understanding the present Java Memory Model is the draft journal paper [7]. During the development of the present memory model, many examples have been discussed on the mailing list, as well as in the test cases [8, 9].

Acknowledgements. The authors enjoyed discussions on some of the examples in this paper with M. Huisman, G. Petri and P. Cenciarelli. The second author is supported by a PhD studentship awarded by the UK EPSRC, grant EP/C537068. Both authors also acknowledge the support of the EU project Mobius (IST-15905).

References

[1] David Aspinall and Jaroslav Sevcik. Formalizing Java's data race free guarantee. To appear in Theorem Proving in Higher-Order Logics, 2007 (TPHOLs 07). See the web page at `http://groups.inf.ed.ac.uk/request/jmmtheory/`., 2007.

[2] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *16th ESOP*, 2007.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*, chapter Memory Model, pages 557–573. Addison-Wesley Professional, July 2005.

[4] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[5] Jeremy Manson. *The Java memory model*. PhD thesis, University of Maryland, College Park, 2004.

[6] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[7] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. Draft journal paper, 2005.

[8] William Pugh and Jeremy Manson. Java memory model causality test cases, 2004. `http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html`.

[9] William Pugh and Jeremy Manson. Java memory model mailing list, 2005. `http://www.cs.umd.edu/~pugh/java/memoryModel/archive/`.

A Precise JMM definitions

The following definitions correspond to those in [3, 6], but are mildly reformulated to match the way we have studied them in [1]. We use $\mathcal{T}$ for the set of thread identifiers, ranged over by t; $\mathcal{M}$ for synchronisation monitor identifiers, ranged over by m; $\mathcal{L}$ for variables (i.e., memory locations), ranged over by v (in examples, `x`, `y`, etc.); and $\mathcal{V}$ for values. The starting point is the notion of *action*.

Definition 1 (Action) *An* action *is a memory-related operation; each action has the following properties: (1) Each action belongs to one thread, denoted $T(a)$. (2) An action has one of the following* action kinds*:*

- volatile read *of* $v \in \mathcal{L}$,
- volatile write *to* $v \in \mathcal{L}$,
- lock *on monitor* $m \in \mathcal{M}$,
- unlock *on monitor* $m \in \mathcal{M}$,
- normal read *from* $v \in \mathcal{L}$,
- normal write *to* $v \in \mathcal{L}$,
- external action.

An action kind includes the associated variable or monitor. The volatile read, write, lock and unlock actions are called synchronisation actions*.*

Definition 2 (Execution) *An* execution $E = \langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$*, where:*

- $A \subseteq \mathcal{A}$ *is a set of actions,*
- P *is a program, which is represented as a function that decides validity of a given sequence of action kinds with associated values if the action kind is a read or a write,*
- *the partial order* $\leq_{po} \subseteq A \times A$ *is the program order,*
- *the partial order* $\leq_{so} \subseteq A \times A$ *is the synchronisation order,*
- $W \in \mathcal{A} \Rightarrow \mathcal{A}$ *is a* write-seen *function. It assigns a write to each read action from A, the $W(r)$ denotes the write seen by r, i.e. the value read by r is $V(W(r))$. The value of $W(a)$ for non-read actions a is unspecified,*
- $V \in \mathcal{A} \Rightarrow \mathcal{V}$ *is a* value-written *function that assigns a value to each write from A, $V(a)$ is unspecified for non-write actions a.*

Definition 3 (Synchronizes-with) *In an execution with synchronisation order $\leq_{so}$, an action a* synchronises-with *an action b (written $a <_{sw} b$) if $a \leq_{so} b$ and a and b satisfy one of the following conditions:*

- *a is an unlock on monitor m and b is a lock on monitor m,*
- *a is a volatile write to v and b is a volatile read from v.*

Definition 4 (Happens-before) *The* happens-before *order of an execution is the transitive closure of the composition of its synchronises-with order and its program order, i.e. $\leq_{hb} = (<_{sw} \cup \leq_{po})^+$.*

Definition 5 (Sequential validity) *We say that a sequence s of action kind-value pairs is* sequentially valid *with respect to a program P if $P(s)$ holds.*

Definition 6 (Well-formed execution) *We say that an* execution $\langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ *is* well-formed *if*

1. $\leq_{po}$ *restricted on actions of one thread is a total order,* $\leq_{po}$ *does not relate actions of different threads.*
2. $\leq_{so}$ *is total on synchronisation actions of* A.
3. $\leq_{so}$ *is an omega order, i.e.* $\{y \mid y \leq_{so} x\}$ *is finite for all* x.
4. $\leq_{so}$ *is consistent with* $\leq_{po}$, *i.e.* $a \leq_{so} b \wedge b \leq_{po} a \Longrightarrow a = b$.
5. W *is properly typed: for every non-volatile read* $r \in A$, $W(r)$ *is a non-volatile write; for every volatile read* $r \in A$, $W(r)$ *is a volatile write.*
6. *Locking is proper: for all lock actions* $l \in A$ *on monitors* m *and all threads* t *different from the thread of* l, *the number of locks in* t *before* l *in* $\leq_{so}$ *is the same as the number of unlocks in* t *before* l *in* $\leq_{so}$.
7. *Program order is intra-thread consistent: for each thread* t, *the sequence of action kinds and values*[5] *of actions performed by* t *in the program order* $\leq_{po}$ *is sequentially valid with respect to* P.
8. $\leq_{so}$ *is consistent with* W: *for every volatile read* r *of a variable* v *we have* $W(r) \leq_{so} r$ *and for any volatile write* w *to* v, *either* $w \leq_{so} W(r)$ *or* $r \leq_{so} w$.
9. $\leq_{hb}$ *is consistent with* W: *for all reads* r *of* v *it holds that* $r \not\leq_{hb} W(r)$ *and there is no intervening write* w *to* v, *i.e. if* $W(r) \leq_{hb} w \leq_{hb} r$ *and* w *writes to* v *then*[6] $W(r) = w$.

Definition 7 (Legality) *A well-formed execution* $\langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ *with happens before order* $\leq_{hb}$ *is* legal *if there is a "committing" sequence of sets of actions* C_i *and well-formed "justifying" executions* $E_i = \langle A_i, P, \leq_{po_i}, \leq_{so_i}, W_i, V_i\rangle$ *with happens-before* $\leq_{hb_i}$ *and synchronises-with* $<_{sw_i}$, *such that* $C_0 = \emptyset$, $C_{i-1} \subseteq C_i$ *for all* $i > 0$, $\bigcup C_i = A$, *and for each* $i > 0$ *the following rules are satisfied:*

1. $C_i \subseteq A_i$.
2. $\leq_{hb_i} |_{C_i} \;=\; \leq_{hb} |_{C_i}$.
3. $\leq_{so_i} |_{C_i} \;=\; \leq_{so} |_{C_i}$.
4. $V_i|_{C_i} = V|_{C_i}$.
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$.
6. *For all reads* $r \in A_i - C_{i-1}$ *we have* $W_i(r) \leq_{hb_i} r$.
7. *For all reads* $r \in C_i - C_{i-1}$ *we have* $W_i(r) \in C_{i-1}$ *and* $W(r) \in C_{i-1}$.

[5] The *value* of an action a is $V(a)$ if a is a write, $V(W(a))$ if a is a read, or an arbitrary value otherwise.

[6] The Java Specification omits the part "$W(r) = w$", which is clearly wrong since happens-before is reflexive.

8. *Let's denote the edges in the transitive reduction of $\leq_{hb_i}$ without all edges in $\leq_{po_i}$ by $<_{ssw_i}$. We require that if $x <_{ssw_i} y \leq_{hb_i} z$ and $z \in C_i - C_{i-1}$, then $x <_{sw_j} y$ for all $j \geq i$.*

9. *If x is an external action, $x \leq_{hb_i} y$, and $y \in C_i$, then $x \in C_i$.*

Note that although the definition of legality does not mention the term "well-behaved execution" directly, rule 6 of legality ensures that the first justifying execution E_1 is well-behaved.

Definition 8 (Sequential consistency) *An execution is* sequentially consistent *if there is a total order consistent with the execution's program order and synchronisation order such that every read in the execution sees the most recent write in the total order.*

Definition 9 (Conflict) *An execution has a* conflicting *pair of actions a and b if both access the same variable and either a and b are writes, or one of them is a read and the other one is a write.*

Definition 10 (DRF) *A program is* data race free *if in each sequentially consistent execution of the program, for each conflicting pair of actions a and b in the execution we have either $a \leq_{hb} b$ or $b \leq_{hb} a$.*